# Chapter 1: Tic Tac Toe

Gabe Sekeres

February 18, 2025

Cornell University

# Baseline Code

## Languages

- There are advantages and disadvantages to any coding language

---

[1]Matters a lot for LLMs!

## Languages

- There are advantages and disadvantages to any coding language
- Python: extremely well-known, intuitive, better documented than anything else,[1] basically industry standard

---

[1]Matters a lot for LLMs!

## Languages

- There are advantages and disadvantages to any coding language
- Python: extremely well-known, intuitive, better documented than anything else,[1] basically industry standard
- Julia: none of these things, but extremely fast

---

[1]Matters a lot for LLMs!

1

## Languages

- There are advantages and disadvantages to any coding language
- Python: extremely well-known, intuitive, better documented than anything else,[1] basically industry standard
- Julia: none of these things, but extremely fast
- Zhang code: Python takes 37.33 seconds, Julia takes 3.29, and it reduces to 2.53 without printing

---

[1]Matters a lot for LLMs!

## Languages

- There are advantages and disadvantages to any coding language
- Python: extremely well-known, intuitive, better documented than anything else,[1] basically industry standard
- Julia: none of these things, but extremely fast
- Zhang code: Python takes 37.33 seconds, Julia takes 3.29, and it reduces to 2.53 without printing
- I use Julia. (I'll present the Python, it's more readable)

---

[1]Matters a lot for LLMs!

1

## Classes[2]

Classes are the objects of interest in this code. Each class has a number of inherent attributes, and essentially acts as a tuple of those attributes. We have four:

1. State $= \langle \text{data}, \text{winner}, \text{hash}, \text{end} \rangle$. This is a single board.
2. Player $= \langle \hat{V}(\mathcal{S}), \alpha, \varepsilon, \text{states}, \text{greedy}, \{X, O\} \rangle$. This contains all the parameters you'd expect, as well as two vectors, states and greedy.
3. Judger $= \langle P_1, P_2, \text{currentPlayer}, \text{currentState} \rangle$. This class runs the game. More on it later.
4. HumanPlayer $= \langle \{X, O\}, \text{state} \rangle$. This doesn't matter for us, I will be ignoring all of the human parts from here on.

---

[2](in Julia, struct)

## Class Functions

Each class has a set of functions inherent to it. They only use that class's elements. In Python, they are defined in the class, in Julia you define them elsewhere. Each class has a constructor function, where you call *e.g.* State() to build a state.

**Remark.** Best practice is functional programming, for speed, modularity, and readability. The vast majority of what we'll do is defining functions of different types

## State Functions - hash(State)

- Take in a certain state, and assign it a unique number (the hash value)

```python
def hash(self):
    if self.hash_val is None:
        self.hash_val = 0
        for i in np.nditer(self.data):
            self.hash_val = self.hash_val * 3 + i + 1
    return self.hash_val
```

## State Functions - is_end(State) Part 1

- Take in a board, check if the game is over, if so add the winner
  and end value to the state, otherwise add false for both.

```python
def is_end(self):
    if self.end is not None:
        return self.end
    results = []
    # check row
    for i in range(BOARD_ROWS):
        results.append(np.sum(self.data[i, :]))
    # check columns
    for i in range(BOARD_COLS):
        results.append(np.sum(self.data[:, i]))
    # check diagonals
    trace = 0
    reverse_trace = 0
    for i in range(BOARD_ROWS):
        trace += self.data[i, i]
        reverse_trace += self.data[i, BOARD_ROWS - 1 - i]
    results.append(trace)
    results.append(reverse_trace)
```

## State Functions - is_end(State) Part 2

```python
for result in results:
    if result == 3:
        self.winner = 1
        self.end = True
        return self.end
    if result == -3:
        self.winner = -1
        self.end = True
        return self.end

# whether it's a tie
sum_values = np.sum(np.abs(self.data))
if sum_values == BOARD_SIZE:
    self.winner = 0
    self.end = True
    return self.end

# game is still going on
self.end = False
return self.end
```

## State Functions - next_state(State, i, j, symbol)

- Add a move to the board

```python
def next_state(self, i, j, symbol):
    new_state = State()
    new_state.data = np.copy(self.data)
    new_state.data[i, j] = symbol
    return new_state
```

## Global State Functions

- These get all of the states that are possible to attain from gameplay:

```python
def get_all_states():
    current_symbol = 1
    current_state = State()
    all_states = dict()
    all_states[current_state.hash()] = (current_state, current_state.is_end())
    get_all_states_impl(current_state, current_symbol, all_states)
    return all_states
```

```python
def get_all_states_impl(current_state, current_symbol, all_states):
    for i in range(BOARD_ROWS):
        for j in range(BOARD_COLS):
            if current_state.data[i][j] == 0:
                new_state = current_state.next_state(i, j, current_symbol)
                new_hash = new_state.hash()
                if new_hash not in all_states:
                    is_end = new_state.is_end()
                    all_states[new_hash] = (new_state, is_end)
                    if not is_end:
                        get_all_states_impl(new_state, -current_symbol,
                            all_states)
```

## Player Functions - reset(Player), set_state(Player, State)

- Return the player to the beginning of the game, resetting the attained states and their respective choices

```python
def reset(self):
    self.states = []
    self.greedy = []
```

- Add a State to the list, with a greedy choice

```python
def set_state(self, state):
    self.states.append(state)
    self.greedy.append(True)
```

## Player Functions - set_symbol(Player, symbol)

- Add a symbol ($\{1, -1\} \equiv \{X, O\}$), and the initial estimations

```python
def set_symbol(self, symbol):
    self.symbol = symbol
    for hash_val in all_states:
        state, is_end = all_states[hash_val]
        if is_end:
            if state.winner == self.symbol:
                self.estimations[hash_val] = 1.0
            elif state.winner == 0:
                # we need to distinguish between a tie and a lose
                self.estimations[hash_val] = 0.5
            else:
                self.estimations[hash_val] = 0
        else:
            self.estimations[hash_val] = 0.5
```

## Player Functions - backup(Player)

- After each game, update the estimations using TD learning

```python
def backup(self):
    states = [state.hash() for state in self.states]

    for i in reversed(range(len(states) - 1)):
        state = states[i]
        td_error = self.greedy[i] * (
            self.estimations[states[i + 1]] - self.estimations[state]
        )
        self.estimations[state] += self.step_size * td_error
```

Math:
$$V(S_t) = V(S_t) + \alpha\Big[V(S_{t+1}) - V(S_t)\Big]$$

## Player Functions - act(Player)

- Choose an action, based on rand() $\sim$ Uniform[0, 1) and $\varepsilon$

```python
def act(self):
    state = self.states[-1]
    next_states = []
    next_positions = []
    for i in range(BOARD_ROWS):
        for j in range(BOARD_COLS):
            if state.data[i, j] == 0:
                next_positions.append([i, j])
                next_states.append(state.next_state(
                    i, j, self.symbol).hash())
    if np.random.rand() < self.epsilon:
        action = next_positions[np.random.randint(len(next_positions))]
        action.append(self.symbol)
        self.greedy[-1] = False
        return action
    values = []
    for hash_val, pos in zip(next_states, next_positions):
        values.append((self.estimations[hash_val], pos))
    # to select one of the actions of equal value at random due to Python's
        sort is stable
    np.random.shuffle(values)
    values.sort(key=lambda x: x[0], reverse=True)
    action = values[0][1]
    action.append(self.symbol)
    return action
```

## Player Functions - save_policy(Player) / load_policy(Player)

- Save the estimations and load them later. I did this very differently, where I had the train function return the converged expectations and kept them as variables. I don't understand why they did it like this.

```python
def save_policy(self):
    with open('policy_%s.bin' % ('first' if self.symbol == 1 else 'second'),
            'wb') as f:
        pickle.dump(self.estimations, f)

def load_policy(self):
    with open('policy_%s.bin' % ('first' if self.symbol == 1 else 'second'),
            'rb') as f:
        self.estimations = pickle.load(f)
```

## The Judger Class

- Why does this exist?
- Short answer: functional programming
- Long answer: It's significantly easier to put this all in a different class, rather than having to alternate the players / define the parameters of the game manually.
- Tldr: It's slightly annoying but makes it easier to change the game. See Extensions below.

## Judger Functions - reset(Judger) / alternate(Judger)

- Reset the two players

```
def reset(self):
    self.p1.reset()
    self.p2.reset()
```

- Switch who plays *X* and who plays *O*

```
def alternate(self):
    while True:
        yield self.p1
        yield self.p2
```

## Judger Functions - play(Judger)

- Run a single iteration of the game

```python
def play(self):
    alternator = self.alternate()
    self.reset()
    current_state = State()
    self.p1.set_state(current_state)
    self.p2.set_state(current_state)
    while True:
        player = next(alternator)
        i, j, symbol = player.act()
        next_state_hash = current_state.next_state(i, j, symbol).hash()
        current_state, is_end = all_states[next_state_hash]
        self.p1.set_state(current_state)
        self.p2.set_state(current_state)
        if is_end:
            return current_state.winner
```

## Global Functions - train(epochs, print_every_n=500)

- Play $N =$ epochs games, printing results every 500 iterations. Have each player learn after each game.

```python
def train(epochs, print_every_n=500):
    player1 = Player(epsilon=0.01)
    player2 = Player(epsilon=0.01)
    judger = Judger(player1, player2)
    player1_win = 0.0
    player2_win = 0.0
    for i in range(1, epochs + 1):
        winner = judger.play(print_state=False)
        if winner == 1:
            player1_win += 1
        if winner == -1:
            player2_win += 1
        if i % print_every_n == 0:
            print('Epoch %d, player 1 winrate: %.02f, player 2 winrate: %.02f'
                % (i, player1_win / i, player2_win / i))
        player1.backup()
        player2.backup()
        judger.reset()
    player1.save_policy()
    player2.save_policy()
```

## Global Functions - compete(turns)

- Play for `turns` games, where each player is always greedy

```python
def compete(turns):
    player1 = Player(epsilon=0)
    player2 = Player(epsilon=0)
    judger = Judger(player1, player2)
    player1.load_policy()
    player2.load_policy()
    player1_win = 0.0
    player2_win = 0.0
    for _ in range(turns):
        winner = judger.play()
        if winner == 1:
            player1_win += 1
        if winner == -1:
            player2_win += 1
        judger.reset()
    print('%d turns, player 1 win %.02f, player 2 win %.02f' % (turns,
        player1_win / turns, player2_win / turns))
```

## Actual Code

- Now that we have all the functions, this is all we need:

```
import numpy as np
import pickle

BOARD_ROWS = 3
BOARD_COLS = 3
BOARD_SIZE = BOARD_ROWS * BOARD_COLS
all_states = get_all_states()

train(int(1e5))
compete(int(1e3))
play()
```

# Extensions

## Questions

1. What happens if a tie is treated as a loss?
2. (Marco) What if we start at the analytic solution (henceforth, minimax estimations)?
3. What happens if we start with random estimations?
4. What happens if we train against a random player rather than a reinforcement learner?

## Models

- Since I used Julia, I was able to train for 3m epochs (with $\varepsilon = 0.1$) fairly easily. I initialized five different learners:
  1. Baseline, win worth 1, tie worth 0.5, loss worth 0
  2. Minimax, where we start at the minimax estimations
  3. Random, starting at random estimations
  4. No Ties, Baseline but ties worth 0
  5. Against Random, Baseline but trained against a random mover

## Competition Results

- I played the models against each other for 10,000 rounds
- Baseline, Minimax, No Ties, and Against Random all were able to attain 100% tie rate (except when No Ties was playing $O$, of course)
- Random, where we drew each initial estimation from Uniform$[0, 1]$, lost every game against the other models
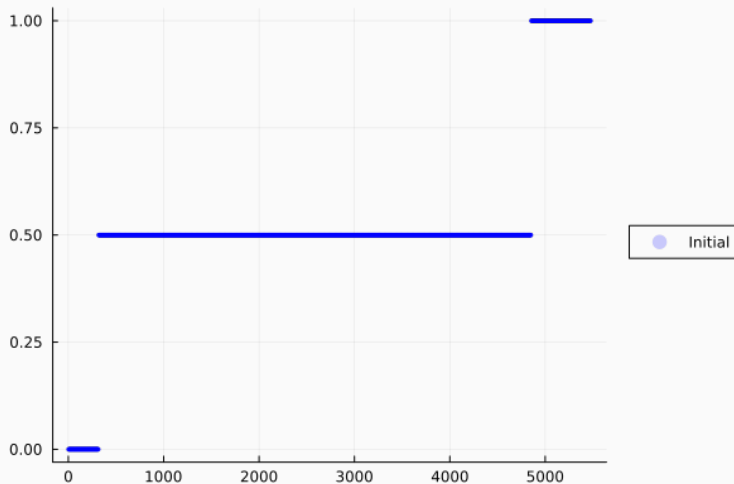
# Estimation Differences ($X$ Player)
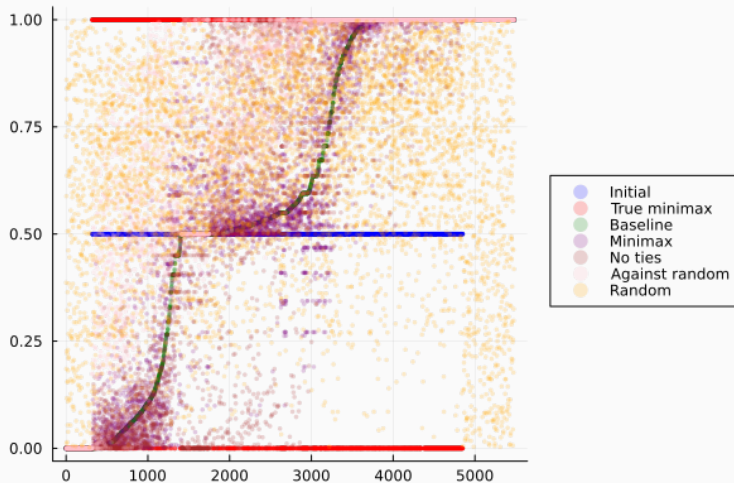
## Estimation Differences (*O* Player)